DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System programming

Organización del Computador II

Segundo Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Leandro Ezequiel Vigali	951/12	levleandro@gmail.com
Franco Sebastián Liza	258/16	franco.s.liza@gmail.com
Patricio Sosa	218/16	patriciososa91@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Introducción

El objetivo de este Trabajo Práctico es realizar una emulación completa de un sistema IBM PC junto con sus operaciones a más bajo nivel que realiza la computadora, en particular definir una tabla de descriptores, pasar a modo protegido, interrupciones, habilitar paginación, tareas, ETC.

2. Desarrollo

Comentario: todas las etiquetas están definidas en `defines.h` para verificar su valor correspondiente. Puede que algunas funciones explicadas en este informe sufran algún cambio para la entrega final, esto es porque seguramente necesitemos una funcionalidad o ya no sea necesaria, además de que el código a testear y hacer en cada ejercicio posiblemente no lo entreguemos ya que algunos no son vitales para el sistema y así limpiamos un poco nuestro código.

2.1. Ejercicio 1

En este punto necesitamos crear los segmentos necesarios para que el sistema funcione con segmentación flat(segmentos solapadas de 0 a X memoria) y qué espacios de memoria direccionamos, pero antes de ello vamos a dar una breve descripción de la GDT.

La GDT está definida como:

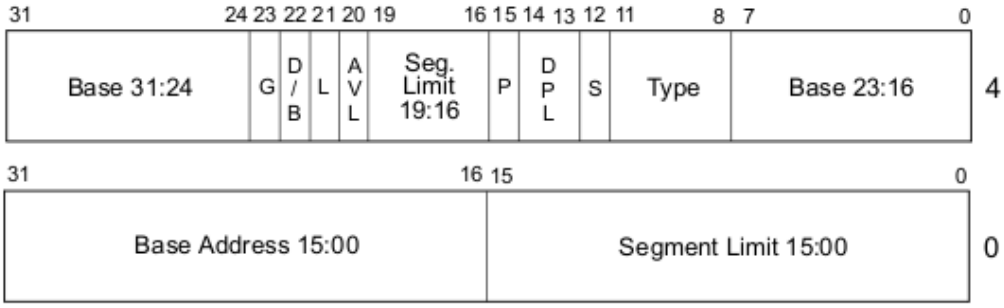


Figura 1: La gdt y sus entradas

En particular vamos a definir 4 segmentos en la GDT, 2 de código y 2 de datos y uno reservado para el sistema el cual es el segmento nulo. La cátedra nos provee la estructura de la GDT y se encuentra en `gdt.h`.

Vamos a utilizar los índices del 10 para arriba ya que los primeros 9 ya se encuentran en uso.

Como tenemos que direccionar los primeros 201MB de memoria ya nos damos una idea de que el campo base de cada uno de los segmentos a definir es 0. Y en el campo límite seteamos los 201MB, y **g** = 1 (granularity) ya que sino no podríamos acceder a dicho espacio, es decir la memoria está segmentada en bloques de 4kb. Para el límite la cuenta que hacemos para pasarlo a hexadecimal, es pasar los 201MB a KB, eso son 205824KB pues 1MB = 1024MB, luego lo dividimos por 4 que nos dará el número de bloques en 4k de nuestra memoria, en este caso serán 51456, como direccionamos hasta el último byte, le restamos 1 y lo pasamos a hexadecimal, en este caso la dirección será: `0xC8FF` como limite. Estos segmentos, los cuatro, están solapados están todos en el mismo lugar, pero en este caso no nos hace problema pues trabajaremos con segmentación flat.

Para los segmentos de nivel 0 tendrán el campo **dpl** en 0 y para los de nivel 3 dicho campo será el 3. A los segmentos de datos tanto nivel 0 como 3, le asignamos en el campo **type** 0x02 para que el segmento sea tanto de lectura como de escritura. Para los segmentos de código el campo **type** será 0x0A el cual se refiere a que estos segmentos serán de lectura y ejecutables.

Por último el campo **d/b** en cada segmento estará en 1, pues son segmentos de 32 bits. Veamos solo un ejemplo para mostrar los descriptores de código y datos de nivel 0 el cual describe lo mencionado en el párrafo de arriba, el descriptores se pueden encontrar en gdt.c .

```
    [GDT_CODE_0] =
      {
            .limit_15_0 = 0xC8FF,   /* limit[0:15]  */
            .base_15_0 = 0x0000,    /* base[0:15]   */
            .base_23_16 = 0x00,     /* base[23:16]  */
            .type = 0xA,            /* type         */
            .s = 0x01,              /* s            */
            .dpl = 0x00,            /* dpl          */
            .p = 0x01,              /* p            */
            .limit_19_16 = 0x00,    /* limit[16:19] */
            .avl = 0x0,             /* avl          */
            .l = 0x0,               /* l            */
            .db = 0x1,              /* db           */
            .g = 0x01,              /* g            */
            .base_31_24 = 0x00,     /* base[31:24]  */
      },
    [GDT_DATA_0] =
      {
            .limit_15_0 = 0xC8FF,   /* limit[0:15]  */
            .base_15_0 = 0x0000,    /* base[0:15]   */
            .base_23_16 = 0x00,     /* base[23:16]  */
            .type = 0x2,            /* type         */
            .s = 0x01,              /* s            */
            .dpl = 0x00,            /* dpl          */
            .p = 0x01,              /* p            */
            .limit_19_16 = 0x00,    /* limit[16:19] */
            .avl = 0x0,             /* avl          */
            .l = 0x0,               /* l            */
            .db = 0x1,              /* db           */
            .g = 0x01,              /* g            */
            .base_31_24 = 0x00,     /* base[31:24]  */
      },
```

Lo siguiente es completar todo lo necesario para pasar a modo protegido. Ahora nos situaremos en kernel.asm , y lo que haremos es habilitar A20 para poder direccionar más de 20 bits.

Dicho esto tenemos que levantar de memoria la estructura GDT_DESC, el cual se refiere al descriptor de segmento que tiene todas nuestras entradas definidas anteriormente. Para ello podremos leerlo con la instrucción **lgdt** (*load global descriptor table*). Entonces vamos a levantar la GDT de la siguiente manera: lgdt [GDT_DESC] ya que GDT_DESC es un puntero a la estructura de GDT . Una vez hecho esto necesitamos setear el bit menos significativo del cr0, el cual tiene como funcionalidad activar modo protegido para poder trabajar con la segmentación definida(que es flat) y trabajar en 32 bits.

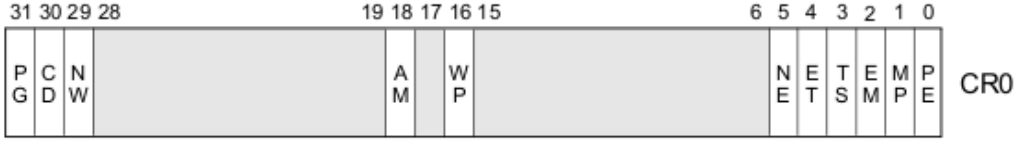


Figura 2: Necesitamos setear PE en 1 para activar modo protegido

Como los registros de control no son registros de propósito general necesitamos realizar una copia de ellos a un registro, por ejemplo eax para luego modificarlo y pegarlo nuevamente en **cr0**.

```
        mov eax, cr0
        or eax, 1
        mov cr0, eax
```

Una vez hecho esto estamos listos para pasar a modo protegido, pero para ello tenemos que cambiar de segmento y lo hacemos con la instrucción **jmp**, en particular un jmp far que se conforma de un segmento y un offset, el cual este último será una etiqueta más abajo en el código, lo que realmente nos importa es cambiar de segmento y en particular al segmento 10, en cual en nuestro caso es el de código de nivel 0. Lo realizamos con jmp ($10 \ll 3$) : modo_protegido , donde la etiqueta **modo_protegido** se encuentra inmediatamente abajo del **jmp**. Por último seteamos los registros de segmentos ss, ds, fs, gs y es con el segmento de datos de nivel 0. Para completarlo tenemos que hacer lo mismo que hicimos con el registro de control cr0, ya que estos no pueden ser modificados por inmediatos, sólo desde otro registro.

```
    xor eax, eax
    mov ax, (10 << 3)
    mov ds, ax
    mov es, ax
    mov ss, ax
    mov gs, ax
    mov fs, ax
```

Continuado con el ejercicio y con modo protegido activado hasta este punto lo siguiente a realizar es crear un segmento de vídeo, el cual según el enunciado se encuentra en la dirección **0xB8000** que describe el área de la pantalla, la pantalla está comprendida por una área de 50 filas y 80 columnas, donde a cada posición le corresponden 2 bytes, un byte para el carácter ASCII de 8 bits y otro byte para los atributos de color frontal y fondo. Por lo que el tamaño del segmento sería 50x80x2 bytes = 8000 bytes y 8000 es **0x1F40**, por lo que el límite del segmento es **0x1F3F**. La granularidad en ese caso es 0, es decir tiene una granularidad de a byte. Este segmento es de datos de tipo lectura y escritura, con un dpl en 0, para que sólo sea accedido por el kernel. Y con esto damos por concluido la realización del ejercicio 1.

2.2. Ejercicio 2

En este punto trabajaremos con la IDT, sus entradas y cómo las definimos.

Definamos primero que nada 20 entradas en la IDT. Las primeras 20, es decir de la entrada 0 a la 19, serán las interrupciones que el procesador generará en base a alguna excepción indicando que problema produjo, algunos son rescatables y se podrá seguir la ejecución pero otros no.

Para ello primero que nada tendremos que completar la IDT con sus atributos, en particular una interrupt gate.

La cátedra nos provee una macro para crear los segmentos, por lo que sólo tenemos que cargar los datos que correspondan, los cuales van a ser el segmento de código de nivel 0, y los atributos.

```
#define IDT_ENTRY(numero)                                                          \
   idt[numero].offset_15_0 = (uint16_t) ((uint32_t)(&_isr ## numero) & (uint32_t) 0xFFFF);      \
   idt[numero].segsel = (uint16_t) (GDT_CODE_0<<3)/*SEL DE SEG DE COD*/;                        \
   idt[numero].attr = (uint16_t) 0x8e00;                                                        \
   idt[numero].offset_31_16 = (uint16_t)((uint32_t)(&_isr ## numero) >> 16 & (uint32_t) 0xFFFF);
```

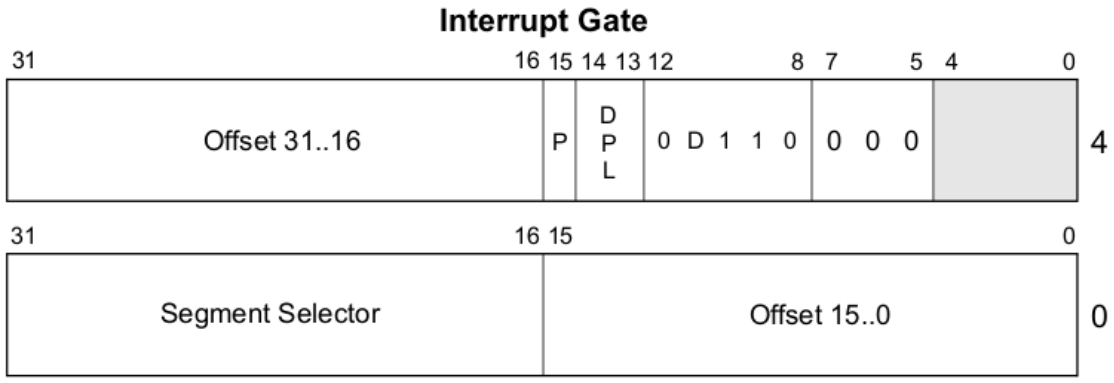


Figura 3: El tipo de descriptor que vamos a utilizar por el momento

Como podemos ver en el código de arriba ya está definido todo, por un lado en **segsel**, completamos con el selector de código de nivel 0 que definimos en el ejercicio 1. En atributos lo describiremos a continuación. podemos ver que dejamos la máscara `0x8E00`, lo cual su representación binaria es:

1000 1110 0000 0000 el cual representa la segunda parte del interrupt gate, en particular los bits 15 a 0.

El bit más significativo se refiere a **p** que nos informa que el segmento se encuentra presente, este mismo se encuentra en 1. los bits que le siguen a izquierda a **p** son **dpl** los cuales son dos y describen el tipo de segmento que es, en este caso 00 ya que es un segmento de acceso sólo del kernel. El próximo campo es **D** que se refiere a si el segmento es de 16 o 32 bits, en este caso es de 32 por lo que está seteado en 1. El resto de campos lo dejamos tal cual como está en la figura 3 y juntando todo eso llegamos a `0x8E00`, el cual es la forma en que intel nos recomienda para que se comporte como interrupt gate hecha y derecha.

Lo siguiente es completar cada entrada en `void idt_init()` Es simple, básicamente es llamar a la macro con el número de cada excepción.

```
void idt_init(){
    IDT_ENTRY(0);
    IDT_ENTRY(1);
    IDT_ENTRY(2);
    ...
    IDT_ENTRY(18);
    IDT_ENTRY(19);
    }
```

Pero acá no terminamos todavía, queda bastante por hacer.

Tenemos que dirigirnos a los archivos `isr.asm` y `isr.h`.

En `isr.asm` tenemos que definir dos cosas. Por un lado tenemos que definir las referencias de cada una de las rutinas de atención de las excepciones, de la 0 a la 19, que luego serán necesarias para su correcta ejecución.

Lo hacemos simplemente cómo:

```
;; Rutina de atención de las EXCEPCIONES
    ISR 0
    ISR 1
    ISR 2
    ...
    ISR 18
    ISR 19
```

Y lo siguiente es definir una macro en asm, la cual ejecutará una función que dará el mensaje de cada excepción que el sistema genere, esta última función más tarde la definiremos en idt.c

Como dicha macro tiene que ser transparente con el sistema, es decir no tiene que modificar ningún registro del mismo, vamos a utilizar las instrucciones **pushad** y **popad** que nos guardan en la pila los registros de propósito general y luego los restaura respectivamente, sin importarme el orden en que los guardo, por último como estamos en un sistema de 32bits para pasar los parámetros tenemos que pushearlo a la pila, lo que tenemos que pushear es el número de la excepción que se produjo y que nos servirá como parámetro de una función de C para poder imprimir la excepción en pantalla.

El siguiente código describe lo escrito arriba:

```
%macro ISR 1
global _isr%1       ; no nos olvidemos de definirla como global

_isr%1:
    pushad
    mov eax, %1
    push eax
    call rutina_de_interrupciones
    add esp, 4
    pop eax
    popad
    iret
%endmacro
```

Ahora vamos al archivo isr.h

Lo que tenemos que hacer es definir cada referencia de cada interrupción creada pero para que C pueda reconocerla, simplemente es:

```
void _isr0();
void _isr1();
void _isr2();
...
void _isr18();
void _isr19();
```

Por último como dijimos antes tenemos que definir la función que imprima cada excepción en la pantalla para que nos informe lo sucedido en el sistema.

Vamos al archivo idt.c Dicha función se llama es: void rutina_interrupción(uint8_t number) el siguiente código es muy grande por lo que no vamos a traerlo al informe, lo que sí vamos a contar de esa función es que es un simple **switch** de C el cual por para numero de excepción producida imprimirá en la pantalla con la función print(... parámetros ...) otorgada por la cátedra. y simplemente eso.

Hasta acá por el momento terminamos con el ejercicio 2 con sus detalles de implementación más importante y relevantes para el informe.

2.3. Ejercicio 3

Para este ejercicio vamos a hacer algo muy parecido al ejercicio anterior lo cual es definir nuevas entradas en la IDT una para el teclado y el clock cada uno con su rutina de interrupción, además de unas 4 entradas adicionales para el sistema, la 88, 89, 100 y 123.

Primero que nada vamos a definir las entradas 88, 89, 100 y 123. Antes de ello vamos a definir una entrada en la GDT para una IDT que es muy parecida a la anterior salvo que como dichas interrupciones

serán generadas por software(syscalls) tendrán un nivel de privilegios más bajo, en este caso es 3, por lo que en el campo DPL como se muestra en la figura 3 tendrá 11 por lo que la dirección antes usada para definir los atributos del segmento, la cual era 0x8E00 para a ser 0xEE00

Por un lado esto ya está, ahora continuemos con la rutina de interrupción del reloj y el teclado. Vamos al archivo isr.h y isr.asm, en este último archivo completamos como global las entradas 88, 89, 100 y 123, además también las interrupciones de reloj y teclado las cuales son la 32 y 33 respectivamente.

```
    global _isr32
    global _isr33
    global _isr88
    global _isr89
    global _isr100
    global _isr123

    y en isr.asm completamos sus referencias:

    void _isr32();
    void _isr33();
    void _isr88();
    void _isr89();
    void _isr100();
    void _isr123();
```

Luego para la interrupción de reloj (supongo que acá va a ir la interrupción final, ver después)

```
_isr32:
    pushad
    call pic_finish1 ; Indica que la interrupcion fue atendida
    call next_clock ; Imprimir el reloj del sistema
    popad
    iret
```

Para que esto sea transparente utilizamos las instrucciones **pushad** y **popad** para que, de manera si hay cambios, no afecte la funcionalidad del sistema al retornar de la interrupción. Luego hacemos un llamado a next_clock proporcionada por la cátedra y a pic_finish1 para avisarle al sistema que terminamos con dicha rutina y de cierta manera *apagamos* el pic.

NOTA: Puede que esta implementación y las siguientes sufran alteraciones para la entrega final agregando o sacando cosas que nos solicitan y las que no son necesarias.

Sigamos con la interrupción de teclado. (-¿Puede dejar de decir interrupción? -NO.)

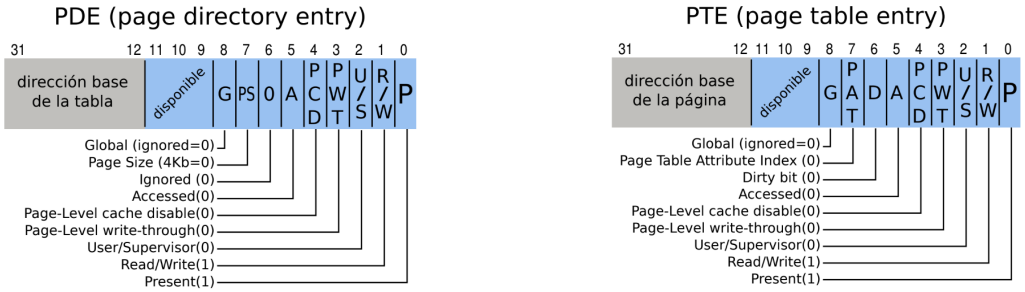
Para ello vamos a definir unos mensajes los cuales serán los números del 0 al 9 para que una vez que se presionen se puedan mostrar en pantalla(use of print) y los **make codes** de las teclas, que casualmente van de 2 a B. Ver `https://stanislavs.org/helppc/make_codes.html` Definimos en isr.asm:

Luego de eso, lo importante es definir la interrupción del teclado la cual también debe ser transparente. Para leer las teclas estaremos leyendo desde el puerto 0x60 del pic, esto dentro de la interrupción con la instrucción **in**. Para simplificar las cosas definimos una función en asm llamada(se podría haber hecho en C lo sé) printScanCode. Concluyendo con el ejercicio 3, el código para esta interrupción es:

```
_isr33:
    pushad
    in al, 0x60 ; Captura una tecla
    push eax
    call printScanCode ;Rutina para Imprimir el ScanCode
    add esp,4
    call pic_finish1
    popad
    iret
```

2.4. Ejercicio 4

En este ejercicio vamos trabajar con las estructuras de memoria y habilitaremos otro nivel de protección de memoria. Para el primer ejercicio inicializar el directorio y tablas de páginas del kernel, esto lo haremos en la función `paddr_t mmu_init_kernel_dir(void)`. A continuación veremos cómo lo definimos en palabras, la implementación de ello puede verse en el el archivo `mmu.c`.
Para lograrlo armamos las estructuras de paginación como se muestra en la Figura 4, tomando la dirección del page directory en el cr3 (0x25000) se puede localizar el directorio de páginas con 1024 entradas, para mapear 4MB establecemos que solo una entrada esté presente, pues cada entrada del directorio contiene 1024 entradas de tablas de páginas y cada una es capaz de referenciar 4KB de memoria física lo que nos devuelve 4MB. Realizamos dos ciclos, el primero para establecer que cada entrada del directorio de páginas y las tablas de páginas no se encuentran presentes ya que en principio podría contener basura,el segundo ciclo se encarga del mapeo, seteando los privilegios de supervisor(PAGS), ya que nos encontramos mapeando un kernel y no buscamos que las tareas sean capaces de leer la memoria destinada al sistema, lectura, escritura y presente, finalmente también se guarda la dirección base de cada 4KB de memoria física dentro de los primeros 4 MB.



(a) Page Directory

(b) Page Table

Figura 4: Estructura de directorio y tablas de páginas

Estas estructuras las definimos en en archivo `mmu.h` las cuales las llamamos **pd_entry** y **pt_entry**.

Luego, el siguiente paso es activar paginación, que básicamente es setear el bit más significativo del cr0, la cual lo podemos ver en la figura 2.
Finalmente el código para activarlo en `kernel.asm` es simplemente:

```
mov eax, cr0
or eax, 0x80000000
mov cr0, eax
```

2.5. Ejercicio 5

Para este ejercicio tendremos que definir el directorio de páginas para el kernel y para las tareas.
Antes que nada inicializamos las estructuras necesarias para la administración de memoria del área libre de kernel las cuales se detallan a continuación.
Definimos una variable global llamada `proxima_pagina_libre` la cual va a llevar el conteo de páginas otorgada por el sistema el cual la primera libre es la cual es 0x100000, (la primera página del área libre del kernel). Esto se hace dentro la función `void mmu_init(void)`.

Luego tenemos la siguiente función `paddr_t mmu_next_free_kernel_page(void)` la cual nos otorgará una nueva página que le solicitamos(de 4KB), incrementando el contador de próxima_pagina_libre. El código es el siguiente.

```
paddr_t proxima_pagina_libre;
void mmu_init(void) {
    proxima_pagina_libre = INICIO_DE_PAGINAS_LIBRES;
}

paddr_t mmu_next_free_kernel_page(void) {
    paddr_t pagina_libre = proxima_pagina_libre;
    proxima_pagina_libre += PAGE_SIZE;
    return pagina_libre;
}
```

Una vez realizado esto, se pide definir una función capaz de mapear a cualquier dirección física una dirección lineal, explicaremos cómo realizarlo pero el código se puede ver en el archivo mmu.c y las funciones implementadas son void mmu_map_page(uint32_t cr3, vaddr_t virt, paddr_t phy, uint32_t attrs) y void mmu_unmap_page(uint32_t cr3, vaddr_t virt) .

Bueno en la primera función mmu_map_page, lo que hacemos es obtener el CR3 ya que tiene el directorio de páginas del kernel, y dada la dirección virtual, vamos a asignarle una dirección física, ambas pasados por parámetros. Dicha traducción lo hacemos obteniendo como lo muestra la siguiente figura.

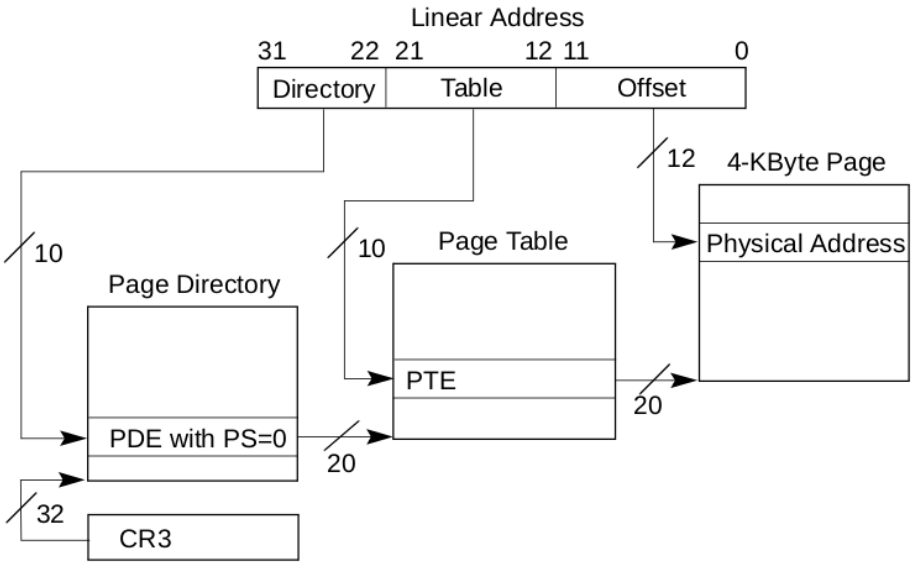


Figura 5: De lineal a física

Lo que haremos primero en la función. Es obtener los OFFSETS correspondientes para indexar sobre las estructuras y en el medio setearle las configuraciones correspondientes.

Luego de esto antes de hacer el mapeo debemos preguntarnos si el directorio de páginas correspondiente al offset del DIRECTORIO es verificar si se encuentra presente, de ser así no haremos nada ya que está mapeada hacia alguna dirección. De no estar mapeada ahí trabajaremos seteando los atributos correspondientes de la nueva entrada del directorio de páginas y también a la nueva entrada al directorio de tablas la cual también debemos dejarla como presente. Es importante que el campo dir_base del directorio de páginas tenga la dirección a la entrada nueva en el directorio de tablas. Esto es pues, para que se realice correctamente la traducción al tratar de acceder con una dirección lógica. Cosas importante para no olvidar, utilizaremos el parámetro attrs, para setear algunos atributos a los directorios, para que sean mapeados páginas con cierta necesidad, ya sean de usuario o supervisor, o que estén protegidos contra escritura o no.

mmu_init_task_dir Esta función se encarga de crear el mapa de memoria para una tarea en particular. Para ello en primer lugar se pide memoria con la función mmu_next_free_kernel_page() para el nuevo directorio de la tarea. Luego dividimos en 2 partes la implementación de este a función que son las siguientes:

1. Kernel: En primer lugar se mapea todo el kernel, es decir los primeros 4MB, con identity mapping usando la función mmu_map_page.

2. Mapeo y copiado de la tarea en cuestión: En primer lugar se realiza el mapeo de la tarea en el cr3 creado para tal, desde la dirección física a su correspondiente virtual. Luego se procede a realizar el copiado de la tarea tal cual como se indica en la figura 6, en la cual se utiliza la tarea RICK como ejemplo. Es decir se cargar el cr3 actual (kernel), y se realiza un mapeo temporal con identity mapping a la dirección que va a ocupar la tarea y se copia el código.

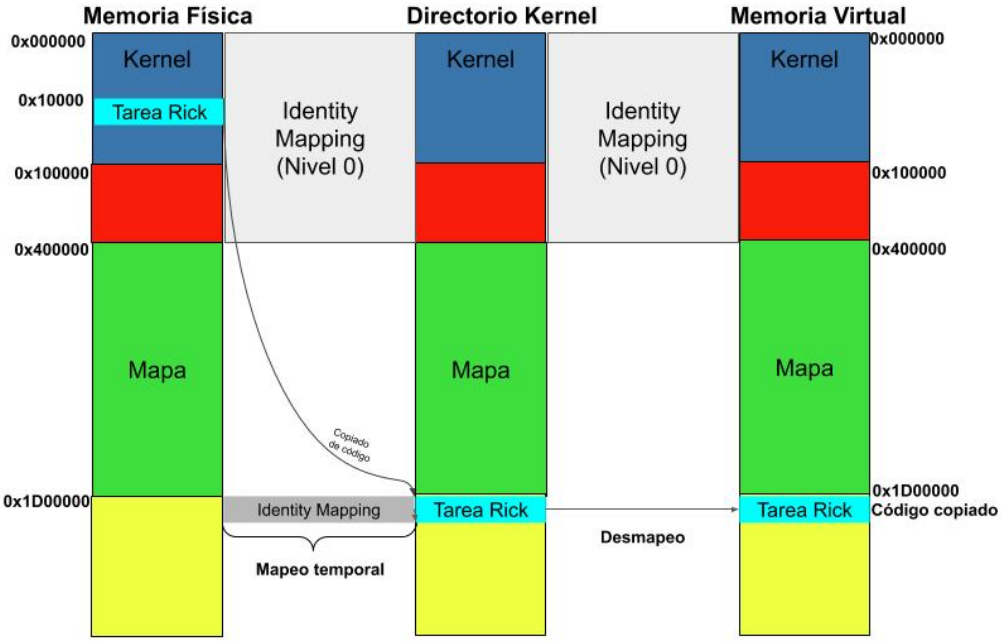


Figura 6: Mapeo temporal, copia y desmapeo para la tarea RICK

2.6. Ejercicio 6

Antes que nada tenemos que definir 2 entradas nuevas en la GDT que son de vital importancia, una entrada para la tarea inicial y otra para la tarea idle. Una vez que hacemos esto saltaremos de la tarea inicial a la idle, la cual mostrará un reloj en la parte inferior derecha de la pantalla, esto simboliza que el sistema está ejecutando dicha tarea.

Para las entradas en la GDT como se muestra en la figura 6, no hay que decir mucho de ellas, ambas tiene como límite 0x0067, esto es por el tamaño de una tss, son de tipo 0x09 es decir, son segmentos de código de sólo ejecución y por último que ambas estén presentes.

Continuando con el ejercicio debemos definir las estructuras de estas 2 tareas, inicial e idle. Vamos a tss.c, ahí están los detalles. Para la tarea inicial lo dejamos en 0, simplemente porque solo lo vamos usar para saltar a la tarea idle y no nos interesa mucho. Es importante que para la tss de la tarea idle, en eip indicarle donde se encuentra el código que va a ejecutar, también su directorio de páginas y tablas, estos son los de kernel, completar sus selectores de segmento, dato que es una tarea de nivel 0, todos sus selectores serán del mismo nivel(pues kernel).

Una vez hecho eso, debemos completar la función void tss_init(void), le estamos indicando la dirección donde se encuentra cada tss en memoria, que es básicamente llenar sus campos casteando el puntero de la función a uint32_t..

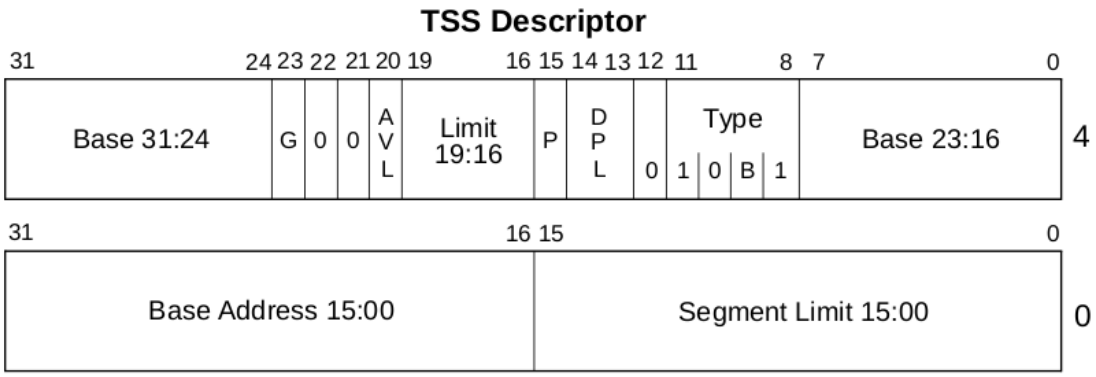


Figura 7: Descriptor de una TSS

```
En tss_init:
    gdt[TSS_IDLE].base_15_0 = ((uint32_t)(&tss_idle) << 16) >> 16;
    gdt[TSS_IDLE].base_23_16 = ((uint32_t)(&tss_idle) << 8) >> 24;
    gdt[TSS_IDLE].base_31_24 = (uint32_t)(&tss_idle) >> 24;

    gdt[TSS_INITIAL].base_15_0 = ((uint32_t)&tss_initial << 16) >> 16;
    gdt[TSS_INITIAL].base_23_16 = ((uint32_t)&tss_initial << 8) >> 24;
    gdt[TSS_INITIAL].base_31_24 = (uint32_t)(&tss_initial) >> 24;
```

Una vez hecho eso, vamos a kernel.asm y hacemos un **call tss_init** para terminar con la inicialización. Finalmente sólo resta completar el código necesario para saltar a la tarea idle desde la tarea inicial. Primero cargamos la tss de la tarea inicial, esto lo hacemos con la instrucción **ltr**

```
    ; Cargar tarea inicial
    mov ax, (TSS_INITAIL << 3)
    ltr ax
```

Finalmente saltamos a la idle, para ello debemos hacerlo con un **jmp far**, pero no es necesario, con un simple **jmp** se puede hacer pues somos el kernel y estamos usando el selector de código de nivel 0.

```
        jmp (TSS_IDLE << 3):0
```

Estructuras de Almacenamiento de las TSS de los MrMeeseeks Para poder acceder a los datos de las TSS creadas para los MrMeeseeks se tiene 2 arreglos de tamaño 10, uno para las tareas creadas por RICK llamada tss_Rickmrms y otro para las tareas creadas por MORTY llamada tss_Mortymrms, las cuales en cada posición poseen el siguiente struct

```
    typedef struct str_MrMs {
        uint8_t in_use;
        tss_t task_seg;
    }tss_mrms;
```

Donde:

- in_use : Indica si la tss esta en uso.
- task_seg: TSS de la tarea correspondiente.

Pilas de nivel 0 Para reutilizar las pilas de nivel 0, utilizadas por los MrMeeseeks, se tiene un arreglo de 20 posiciones llamado pilas_0[] el cual en cada posición almacena una paddr_t.Cada posición representa la pila de nivel 0 de una tarea MrMeeseek, que una vez sacadas del scheduler se guardan en este arreglo para poder reutilizarlas al crear un nuevo MrMeeseeks.

2.7. Ejercicio 7

Para iniciar nuestra estructura de Scheduler tenemos la función sched init en sched.c. Para trabajar con la lista de tareas a ejecutar, nos valdremos de un arreglo global de 23 posiciones llamado sched_task, correspondientes con la cantidad de tareas del sistema. En cada posición del mismo tendremos el siguiente struct:

```
typedef struct str_sched {
    uint8_t is_alive;
    uint16_t tss_selector;
    uint8_t id;
    int pos_x;
    int pos_y;
    int distCel;
    int ticks;
    uint8_t uses_of_gun;
}__attribute__((__packed__, aligned (8))) sched;
```

Donde:

- is_alive: Booleano que indica si una tarea sigue viva. Para las tareas Idle,Rick y Morty se inicializa en TRUE. El resto de las tareas que aun no se activan quedan seteadas en FALSE.
- tss_selector: Indica el selector de TSS de la tarea correspondiente.
- id: Identificador de la tarea
- pos_x: Indica la posición horizontal de la tarea. Puede tomar valores entre 0 y 79.
- pos_y: Indica la posición vertical de la tarea. Puede tomar valores entre 0 y 39.
- distCel: Indica la cantidad máxima de caldas que se puede mover una tarea Mr. Meeseeks cuando utiliza la syscall move, este ítem solo es utilizado para las tareas Mr Messeks, se setea como valor inicial -1 para todas las tareas lo cual indica que está en desuso.
- ticks: Contador de ticks para disminuir la variable distCel, también es únicamente utilizada por las tareas Mr Meeseeks, se setea como valor inicial -1 para todas las tareas lo cual indica que está en desuso.
- uses_of_gun: Indica si la tarea Mr Meeseek ya utilizó o no el arma de portales.

Además, durante la creación del Scheduler, inicializaremos las siguientes variables globales, cuyos propósitos a grandes rasgos es asegurar el correcto comportamiento del juego/sistema:

- current_task : Indica cual es la tarea actual que está corriendo. Inicia con el valor 0, correspondiente a la tarea Idle.
- previous_task : Indica cual fue la tarea que estuvo ejecutándose antes de la tarea actual, para facilitar el cambio de turnos de los jugadores.
- screen_debug: Indica si actualmente en pantalla se muestra el modo debug.
- act_debug:Indica si el modo debug está activado.
- score_rick: Indica el puntaje del jugador RICK.

- score_morty: Indica el puntaje del jugador Morty.

Además de ello también será necesario para poder saltar a la tarea siguiente. En el archivo isr.asm definimos:

```
        sched_task_offset:     dd 0xFFFFFFFF
        sched_task_selector:   dw 0xFFFF
```

Con esto ya tenemos la estructura. Ahora falta que modificar la interrupción de reloj para que salte a la siguiente tarea, pero para ello tenemos que definir la función uint16_t sched_next_task() que se encuentra en sched.c, esta función nos dará el índice en la TSS para cargar su contexto y también su selector de segmento de la tarea a ejecutar. Nuestro caso es *Round Robin* por lo que se estará ejecutando una tarea tras otra de cada jugador por ciclo de reloj.

Luego tenemos que cargar las tareas Rick y Morty. Esto lo hacemos en la función task_init que nos resultó de gran ayuda para simplificar las cosas, donde nos encargamos de hacer sus mapeos, copiar su código y darle un directorio de páginas para cada una de las tareas, además de definir su TSS. Se puede ver con más detalles en tss.c.

```
//Como inicializamos las tareas

void tss_init(void) {
    task_init(&tss_rick,RICK_CODE_PHY,TASK_CODE_VIRTUAL,RICK_CODE,TASK_PAGES);
    task_init(&tss_morty,MORTY_CODE_PHY,TASK_CODE_VIRTUAL,MORTY_CODE,TASK_PAGES);
}
```

```
//Funcion que nos simplfica las cosas, le pasamos todos los parametros necesarios para crear esta tarea.
void task_init(tss_t* new_tss, paddr_t phy_task, vaddr_t virt_task, paddr_t task_code, size_t pages)
{
    *new_tss = (tss_t) {0};
    new_tss->eip = virt_task;
    new_tss->cr3 = mmu_init_task_dir(phy_task,virt_task,task_code,pages);
    new_tss->esp = virt_task + pages*PAGE_SIZE;
    new_tss->ebp = virt_task + pages*PAGE_SIZE;
    new_tss->esp0 = mmu_next_free_kernel_page();
    new_tss->ss0 = GDT_DATA_0 << 3;
    new_tss->cs = (GDT_CODE_3 << 3) + 3;
    new_tss->ds = (GDT_DATA_3 << 3) + 3;
    new_tss->eflags = 0x202;
    new_tss->ss = (GDT_DATA_3 << 3) + 3;
    new_tss->es = (GDT_DATA_3 << 3) + 3;
    new_tss->ds = (GDT_DATA_3 << 3) + 3;
    new_tss->fs = (GDT_DATA_3 << 3) + 3;
    new_tss->gs = (GDT_DATA_3 << 3) + 3;
    new_tss->iomap = 0xFFFF;

    if (pila_0 == 0) {
        new_tss->esp0 = mmu_next_free_kernel_page() + PAGE_SIZE;
    } else {
        new_tss->esp0 = pila_0;
    }
}
```

Con esto ya estamos en condiciones de poder saltar hacia otras tareas. Con todo lo dicho anteriormente, modificamos la _isr32, y continuamos, dejamos el código de como quedo hasta el momento, este

código no es el último por lo que puede surtir modificaciones a la hora de implementar la lógica restante del juego. Luego esparciremos mega semillas por todo el mapa, estas mega semillas también estarán definidos por una estructura, y tendremos un vector con la cantidad total semillas a guardar.

```
_isr32:
    pushad
    call pic_finish1 ; Indica que la interrupción fue atendida
    call next_clock ; Imprimir el reloj del sistema
    call reset_MrMsCel ; ticks de la tareas
    call sched_next_task
    str cx
    cmp ax, cx
    je .fin
    mov [sched_task_selector],ax
    jmp far [sched_task_offset]

    .fin:
    popad
    iret
```

Usamos nuevamente un **jmp far** para cambiar el selector.

Rutinas de excepción Se modificó la macro de las excepciones del sistema para que además de imprimir la excepción que ocurrió, se llama a la rutina **killcurrent_task** para desalojar la tarea que la causó, seteando su campo de struct is_alive en 0 para que no vuelva a ser considera, además libera el slot que ocupa su tss para ser considerado cuando se tenga que crear una nueva tarea, se desmpean sus páginas y se carga la tarea **idle** hasta que se termine el **quantum** de tiempo, luego se procederá a cargar la próxima tarea correspondiente.

Si la tarea que ejecutó la excepción fue Morty o Rick, se termina el juego.

Además se agregó código a la macro, para poder imprimir los datos correspondientes a la tabla de de registros del **modo debug**, y agregando al 4 bytes nulos en la pila de nivel 0 para las excepciones que no contengan código de error, simulando que todas las excepciones tengan un código de error, para facilitar el manejo de estas.

Modo Debug Se modificó la interrupción de teclado para que en el caso de que se apriete la tecla **'y'**, se active/desactive el **modo debug**. Para activar/desactivar el modo debug se agregaron las siguientes funciones.

```
void set_screen_debug(){screen_debug = 1;}

void set_modo_debug() {
    if (act_debug == 1) { // a la espera de una excepcion
        if(screen_debug == 1){ // ya hubo excepcion
            screen_debug = 0;
            reset_screen();
        }else{              // no hubo excepcion
            act_debug = 0;
        }
    }else{      // desactivar debug mode
        act_debug = 1;
    }
}
```

La función **set_modo_debug** se encargará de controlar los estados del modo debug. Como por default el **modo debug** está desactivado, entonces al apretar la tecla **'y'**, la función es llamada y si el **modo debug** está desactivado entonces lo activa, y continua con la ejecución de la tarea que fue interrumpida.

Cuando ocurra una excepción, si el **modo debug** está activo, convocará a la función **imprimir_registros** que se encargará de mostrar por pantalla los valores de los registros antes de que ocurra excepción y la tarea que la ejecutó, activando el campo **screen_debug** en 1 mediante **set_screen_debug**, y el juego se encontrará detenido, ejecutando la tarea **idle**.

Si se vuelve a apretar la tecla **'y'**, entonces se quitará la tabla de debug, pero el modo seguirá activo a la espera de otra excepción. En el caso de que nunca ocurra una excepción y el modo estaba activo, el **modo debug** será desactivado. Y si ocurre una excepción, ocurrirá lo mismo pero el juego no se detendrá y no se mostrará información al respecto por pantalla.

La función **imprimir_registros** obtiene los valores de los registros, stack, etc, por medio de la pila de nivel 0.

Para obtener los valores del stack de nivel 3 de la tarea que causó la excepción se recorre su stack de nivel 0, obteniendo su **esp3**, y se colocan en la pila de nivel 0. Al ser segmentación flat no necesitamos el segmento **ss** de nivel 3.

```
.loop1:
    push ecx
    add edx, 4 ; retorno <- edx = ebp + 4
    push edx
    call virtual_valida
    pop edx
    pop ecx
    cmp eax, 1
    je .valida
.no_valida:
    cmp ecx,7
    je .fin_loop1
    mov DWORD [esp+ecx*4], 0
    inc ecx
    jmp .no_valida
.valida:
    mov eax, [edx]
    mov [esp+ecx*4], eax
    inc ecx
    push ecx
    sub edx, 4 ; edx = ebp
    push edx
    call virtual_valida
    pop edx
    pop ecx
    cmp eax, 1
    jne .no_valida
    mov edx, [edx] ; viejo rbp
    cmp ecx,7
    jne .loop1
.fin_loop1:
```

Para obtener su backtrace se utiliza el ebp, que se conserva a pesar del cambio de privilegio, con el **ebp** y los anteriores, se obtienen las direcciones de retorno y se colocan en la pila de nivel 0.

```
    .loop2:
    push ecx
    push edx
    call virtual_valida
    pop edx
    pop ecx
    cmp eax, 1
    je .valida2
.no_valida2:
    mov DWORD [esp+4*ecx], 0
    inc ecx
    cmp ecx, 6
    je .fin_loop2
    jmp .no_valida2
.valida2:
    mov eax, [edx] ; [esp3]
    mov [esp+4*ecx], eax
    add edx, 4 ; esp3 +4
    inc ecx
    cmp ecx, 6
    jne .loop2
.fin_loop2:
```

En ambos casos se utiliza la función **virtual_valida** para no generar otra excepción al acceder a memoria que no pertenece a las páginas de la tarea. Si la dirección virtual no es válida se colocan ceros.

Además se agregó a la rutina de reloj, la función **check_screen_debug**, para saber si el juego debe pausarse en caso de estar activa la pantalla de debug o el juego debe continuar.

2.8. Ejercicio 8 - La lógica

Gran parte de la lógica a implementar va a estar en el archivo Sched.c , la mayoría de la lógica está implementada sobre C y algunas sobre ASM, donde le daremos el toque final a las interrupciones que faltan.

En sched.c hay muchas funciones que realizan una tarea sencilla, como por ejemplo calcular un módulo, verificar que una posición del mapa es válida(que no esté fuera de rango), una función valor, absoluto, etc, etc. Sobre estas funciones no hablaremos mucho pero si hablaremos de las funciones que ayudan a realizar las interrupciones y las que inicializan y resetean la pantalla.

Empezaremos con la función sched_init la cual va a inicializar todas las tareas, las cuales algunas no están *del todo* inicializadas , es decir tienen definida su TSS y su GDT correspondiente. Las tareas Rick y Morty inician como vivas, el resto de las tareas, las mr meeseeks, las terminamos de inicializar y dejarlas como vivas cuando usamos la int 88. Estas tareas serán guardados en un vector llamado **sched_task** la cual tendrá toda la información de cada tarea y de este vector se podrá consultar sobre la tarea actual con la variable **current_task**, que es la que se estará siendo modificada constantemente por la función **sched_next_task** para ir ejecutando tarea por tarea. También tenemos otra variable **previous_task** que contendrá el identificador de la última tarea que fue desalojada al haber un cambio de tarea.

Luego de esto esparcimos todas las mega semillas en el mapa con la función spread_megaSeeds, para la cual las mega semillas tiene definido el siguiente struct:

```
typedef struct semillas {
    uint32_t position_x;
    uint32_t position_y;
    uint8_t assimilated;
}megaSeeds;
```

La cual tiene toda la información de la semilla, si está asimilada o no, es decir cuando es capturada

por un mr meeseeks y también sus coordenadas sobre el mapa del juego. Toda esta información también estará guardada en un vector aparte llamado **seedsOnMap**, donde la cantidad de semillas a esparcir está definida en defines.h, por la etiqueta **TOTAL_SEEDS**.

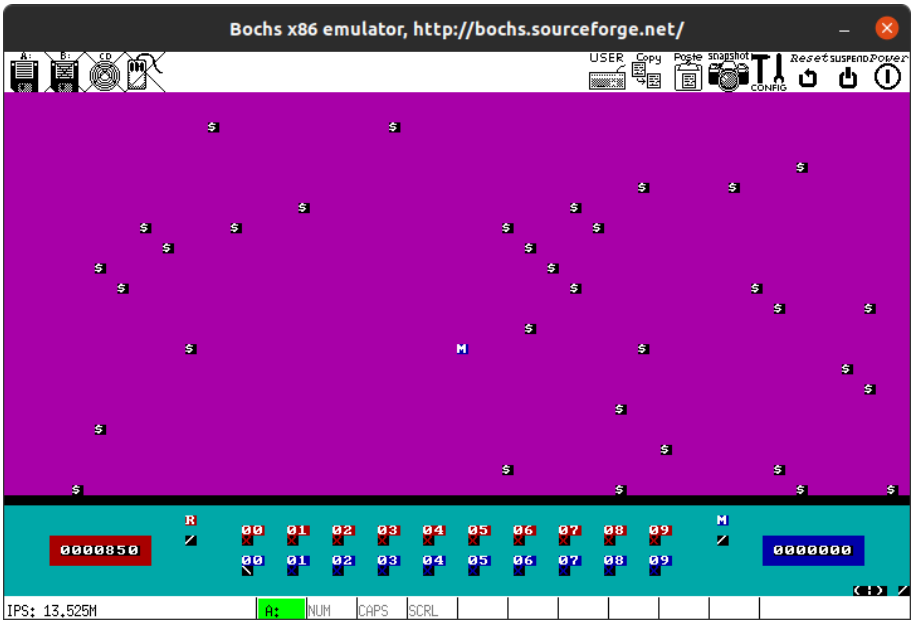


Figura 8: Mega semillas esparcidas en el mapa y una tarea de Morty viva

2.8.1. SERVICIOS DEL SISTEMA

Syscall Meeseeks - INT 88 Se trata del servicio utilizado solamente por Rick o Morty para crear Mr Meeseeks. Se consideran 2 casos:

1. La tarea que llama al servicio es una RICK o una MORTY: A su vez este item se subdivide en 4 casos que se detallan a continuación:

   - Posición Valida: Se verifica que la posición recibida por parámetro sea una posición válida en el mapa. En caso de no ser válida se retorna 0. Para esto se recurre a la función right_postition.
   - Si hay una semilla donde se quiere crear la tarea: Se verifica si en la posición recibida como parámetro si hay una semilla, para esto se recurre a la función move_assimilated(). En caso de encontrar una semilla en la posición no se crea la tarea y se verifica si hay slots disponibles o no recurriendo a la estructura donde almacenamos las tss (tss_Rickmrms o tss_Mortymrms) en cuyos casos se resuelve:
     - Si no hay slots:La función next_tss devolvió -1. Se retorna el valor 0 sin hacer nada.
     - Si hay slots: La función next_tss devolvió el índice de una tss vacía. Se verifica si la tarea es una RICK o una MORTY, se asimila la semilla, se suma puntos al jugador correspondiente se setea la pantalla y se retorna 0.
   - Si la tarea es una Rick:
     - Se verifica si hay slots disponibles en la tss_Rickmrms, en caso de negativo se retorna 0, en caso afirmativo, Se setea la tss in_use en TRUE, se calcula la dirección virtual y física de la tarea a crear, luego se inicializa la tss con esos datos, para ls esp_0 se verifica si hay alguna para reutilizar. Por último se setea los datos correspondiente a la tarea Mr Meeseek a crear en la estructura del scheduler, como is_alive = TRUE, pos_x, pos_y ,distCel = 7, ticks =2. Por último se resetea la pantalla y se devuelve la dirección virtual de dicha tarea.
     - Si no hay slots se retorna 0.
   - Si la tarea es una Morty:

- Se verifica si hay slots disponibles en la tss_Mortymrms, en caso de negativo se retorna 0, en caso afirmativo, Se setea la tss in_use en TRUE, se calcula la dirección virtual y física de la tarea a crear, luego se inicializa la tss con esos datos, para ls esp_0 se verifica si hay alguna para reutilizar. Por último se setea los datos correspondiente a la tarea Mr Meeseek a crear en la estructura del scheduler, como is_alive = TRUE, pos_x, pos_y ,distCel = 7, ticks =2. Por último se resetea la pantalla y se devuelve la dirección virtual de dicha tarea.
- Si no hay slots se retorna 0.

2. La tarea es una MR Meeseek: En este caso como no tienen permitido usar este servicio se llama a la función killcurrent_task(), la cual, al tratarse de una tarea MrMeeseek se encargar de realizar las siguientes acciones:

   - Se setea la estructura del scheduler con los valores por default.
   - Se se pone in_use en FALSE, en la estructura de almacenamiento de TSS.
   - Se accede a dicha TSS para obtener los valores de cr3 y a la pila de nivel 0.
   - Se desmapea las páginas usadas por la tarea, usando la función mmu_unmap_page.
   - Se guarda la pila de nivel 0 para reutilizarla en la estructura pilas_0.

Syscall use portal gun- INT 89 La idea de esta syscall es tomar una tarea del equipo contrario y moverla a algún lugar random del mapa, eso también conlleva remapear su dirección física.

- Lo primero que hacemos es verificar de qué tarea se trata la cual llama al sistema, si es una tarea de Rick o de Morty.
- Una vez hecho eso, verificamos que dicha tarea esté viva, sino está viva entonces vuelvo a tomar un número random, para elegir otra tarea que esté viva, esto siempre vale pues el juego no terminó, es decir hay tareas mr meeseeks de los equipos contrarios.
- Una vez hecho esto, dado que tengo que remapear la tarea contraria debo obtener su CR3, y también debo guardar el CR3 de la tarea que lo invocó, esto es necesario pues se trata de otro directorio.
- Una vez que obtengo la dirección, física y virtual nueva, me preparo a mapearlo y copiar su código. Para ello mapeo la dirección física con identity mapping, esto para usarlo como memoria temporal, luego al mapear la nueva dirección virtual a la nueva dirección física se encontrarán los datos ahí.
- Por último desmapeo el viejo mapeo y el temporal, y mapeo el nuevo directorio, una vez hecho esto restauro el CR3 anterior. Este procedimiento es igual para ambas tareas, salvo que uno u otro caso debo tener unos recaudos.

Syscall look -INT 100 Para esta interrupción definimos una función en C, la cual hace todo el trabajo. Dado que la función que realiza la interrupción es la **current_task** corriente sobre el momento, lo primero que haremos es verificar de qué tarea se trata, si las tareas son Rick o Morty, entonces devolvemos -1 en ambas coordenadas. Luego si se trata de una tarea Mr Meeseek lo que haremos buscar sus coordenadas en el mapa del vector sched_task que tiene toda la info y con respecto a esa posición buscamos las mínimas coordenadas X e Y, más cercanos. Esto es muy fácil, es simplemente es calcular el mínimo de 2 variables y las sumamos para encontrar la distancia Manhattan más corta.

Syscall move -INT 123 Se trata del servicio utilizado solamente por Rick o Morty para crear Mr Meeseeks. Se consideran 4 casos:

1. La tarea que llama al servicio es una RICK o una MORTY: En este caso se mata a la tarea y se salta a la Idle y se termina el juego.
2. La distancia Manhattan es mayor al máximo actual permitido por la tarea en cuestión: En este caso no se desplaza y retorna 0.

3. Si el desplazamiento implica pisar una semilla: Una vez calculada la nueva posición de la tarea, se verifica si hay una semilla en esa posición usando la función move_assimilated, si existe una semilla en dicha posición se realizan las siguientes acciones:
   - Se verifica si la tarea MrMeeseek creada por RICK o MORTY.
   - Se suma los puntos correspondiente al jugador.
   - Se setea la posición de la semilla como asimilada en seedsOnMap.
   - Luego se llama a la función killcurrent_task(), la cual, al tratarse de una tarea MrMeeseek se encargar de realizar las siguientes acciones:
     - Se setea la estructura del scheduler con los valores por default.
     - Se se pone in_use en FALSE, en la estructura de almacenamiento de TSS.
     - Se accede a dicha TSS para obtener los valores de cr3 y a la pila de nivel 0.
     - Se desmapea las páginas usadas por la tarea, usando la función mmu_unmap_page.
     - Se guarda la pila de nivel 0 para reutilizarla en la estructura pilas_0.
4. Se aplica el desplazamiento correspondiente, para lo cual se realizan las siguientes acciones:
   - Se identifica la posición de la tarea
   - Se accede a su TSS, en la estructura tss_Mortymrms o tss_Rickmrms.
   - Se calcula la dirección virtual.
   - Se calcula la nueva dirección física.
   - Se calculan los atributos.
   - Se hace un mapeo temporal en el cr3 actual, usando identity mapping para poder copiar el código de la tarea con la función mmu_map_page.
   - Se copia el código de la tarea desde su dirección virtual a la nueva dirección física.
   - Desmapeo el mapeo temporal que se hizo con la función mmu_unmap_page.
   - Desmapeamos la tarea de su actual dirección física con la función mmu_unmap_page.
   - Mapeamos la nueva dirección física de la tarea con la función mmu_map_page.
   - Seteamos la nueva posición en la estructura del scheduler.
   - Seteamos la pantalla con la función reset_screen.

Aclaración: En cada llamado a algunos de los servicios, antes de volver a correr el código de la tarea en cuestión, se salta a la tarea **idle** hasta completar el quantum restante para continuar con la carga de las demás tareas, la tarea en cuestión retoma su ejecución cuando vuelve a ser su turno.

2.9. Funciones Auxiliares

- `int modulo(int numero, int base)` : Calcula el modulo del numero en la base correspondiente, ambos pasados por parámetro, este siempre dá positivo.
- `int abs(int number)` : Calcula el valor absoluto del número pasado por parámetro. Calcula el valor absoluto del número pasado por parámetro.
- `void screen_init()` : Inicializa la pantalla, imprimiendo el tablero del juego y esparciendo todas las mega semillas por el mapa con la función spread_megaSeeds.
- `void killcurrent_task()` :Se encarga de desalojar la tarea que esta actualmente corriendo. Además según cual sea la tarea que es desalojada se realizan algunas acciones mas que se detallan a continuación.
  1. En primer lugar se setea el valor de sus atributos en el scheduler de la tarea a desalojar. Esta acción se realiza para todas las tareas en general.

2. La tarea a desalojar se trata de una RICK o una MORTY:
3. La tarea a desalojar es una MrMeeseek: En este caso se realiza las siguientes acciones:
   - Se se pone in_use en FALSE, en la estructura de almacenamiento de TSS a la tarea correspondiente.
   - Se accede a dicha TSS para obtener los valores de cr3 y a la pila de nivel 0.
   - Se desmapea las páginas usadas por la tarea, usando la función mmu_unmap_page.
   - Se guarda la pila de nivel 0 para reutilizarla en la estructura pilas_0.

- `int next_tss(tss_mrms *tss_str)` : Toma la estructura que almacena las TSS de los MrMeeseeks y devuelve el índice del array con la siguiente tss que esté habilitada para usar, si no encuentra ninguna TSS disponible retorna -1.
- `paddr_t next_esp0(paddr_t *esp0_str)` : Toma la estructura que almacena todas las pilas de nivel 0 a ser reutilizadas,y retorna la primera pila de nivel 0 distinta de 0, en caso de no encontrar ninguna devuelve 0.
- `bool right_postition(uint32_t pos_x, uint32_t pos_y)` : Verifica si las posiciones recibidas como parámetro, son posiciones válidas dentro del mapa, retorna un valor booleano que representa la verificación.
- `void reset_screen()` : Se encarga de actualizar la pantalla cuando hay algún movimiento, nuevo o algo que altere la pantalla debido a una Syscall.
- `void spread_megaSeeds()` : Esparce las mega semillas por todo el mapa, la cantidad de semillas está definida en defines.h
- `bool move_assimilated(uint32_t pos_x, uint32_t pos_y)` : Verifica si en la posición pasada como parámetro hay una semilla definida, para cual recorre la estructura seedsOnMap, buscando la posición en caso de encontrar dicha posición devuelve TRUE de lo contrario FALSE.
- `search_megaSeeds(uint32_t pos_x, uint32_t pos_y)` : Busca en la estructura seedsOnMap el índice donde se encuentra la semilla pasada como parámetro, en general antes de llamar a esta función se llama a move_assimilated, para chequear si la posición está definida en dicha estructura, aun así la función también devuelve -1, en caso de no estar definida la posición.
- `uint32_t distMan(int desp_x, int desp_y)` : Calcula la distancia Manhattan de las posiciones que toma como parámetro y retorna dicho valor.
- `void imprimir_registros()` : Imprime la tarea, registros, el backtrace y su pila de nivel 3, esta función sirve para imprimir la información del modo_debug.
- `void set_screen_debug()` : Flag de activación de la pantalla del modo_debug.
- `void set_modo_debug()` : Activa/desactiva el modo_debug.
- `uint32_t check_act_debug()` : Devuelve 1 si el modo_debug está activo, 0 sino.
- `uint32_t virtual_valida(uint32_t number)` : Devuelve uno en caso de que la dirección virtual pasado por parámetro se encuentra en el marco de sus páginas de memoria, para no generar un pagefault, devuelve 0 sino.
- `void reset_MrMsCel()` : Se encarga de setear en cada tick de reloj, la estructura del scheduler distCel y ticks, el cual realiza la lógica de las tareas MrMeeseek que consiste reducir en 1 la distancia Manhattan cada 2 ticks de reloj. Para esto recorre el sched_task y verifica si ticks llegó a cero, en ese caso resta 1 a distCel que representa la máxima distancia Manhattan que se puede desplazar un MrMeeseek, una vez que distCel llega a uno se setea tick en -1, para no volver a actualizarlo.